デバイスの記事

SystemCによるビヘイビア合成入門

第4回 ビヘイビア合成可能なC/C++構文

桜井 至

今回は，C/C++の構文の中で，ビヘイビア合成に利用できるものについて解説する．ビヘイビア合成を利用する場合，ポインタを用いた動的な配列宣言は利用できない．また，データ・タイプによって利用可能な演算子とそうでない演算子がある．合成可能な構文をきちんと理解せずに入力記述を作成すると，後から記述の修正が必要になり，余計な手間が増えてしまう． （編集部）

前回までは，ビヘイビア合成可能なSystemC特有の構文について説明してきました．今回は，アルゴリズム記述に使用するビヘイビア合成可能なC/C++構文について紹介します．

ビヘイビア合成可能なC/C++構文サブセットは，OSCI（Open SystemC Initiative）というSystemC普及推進組織の「ビヘイビア合成サブワーキング・グループ」が標準化を進めており，現在（2005年1月），ドラフト版の仕様ができ上がっています．

合成サブセットの仕様は，どのようなC/C++構文がビヘイビア合成可能かの基準を示していますが，C/C++構文からどのようにRTL（register transfer level）記述を生成するかの詳細な手法や構造は定義していません．そのため，実際の実現化手法はビヘイビア合成ツールごとに異なります．

本連載でこれまで紹介したSystemC記述は，OSCIの合成サブセット仕様に基づいています．そのため，基本的にはビヘイビア合成が可能です．しかし，ツールがサポートしている構文の範囲やビヘイビア合成を制御する属性はツールに依存します．詳細な情報については，かならず各ツール・ベンダに問い合わせるようにしてください．

● ファイルはヘッダとメイン・スレッドから成る

まず，簡単な記述例として，FIR（finite impulse response）フィルタのヘッダ・ファイルを**リスト1**に，メイン・スレッド（メイン関数）の記述を**リスト2**に示します．また，この記述例に基づいたFIRフィルタの構造を**図1**に示します．

設計データは一つのファイルにまとめても，複数のファ

リスト1 FIRフィルタのヘッダ・ファイル（fir_filter.h）

```
#ifndef _FIR_FILTER_H_
#define _FIR_FILTER_H_

#include <systemc.h>

#define TAPS 5
typedef sc_uint<8>  UINT8;
typedef sc_uint<16> UINT16;

const sc_uint<4> coefs[TAPS] = {1, 2, 3, 2, 1};

SC_MODULE(fir_filter)
{
  // モジュール入出力宣言
  sc_in<bool>    clk;
  sc_in<bool>    reset;
  sc_in<bool>    input_valid;
  sc_in<bool>    input_mode;
  sc_in<UINT8 >  input_data;

  sc_out<bool>   input_ready;
  sc_out<bool>   output_valid;
  sc_out<UINT8 > output_data;

  // メンバ変数宣言
  UINT8   shift[5];  // 配列宣言
  UINT8   sum_of_coefs;

  // メンバ関数宣言
  void thread0();
  void init(UINT8 data);
  UINT8 calc_fir_filter(UINT8 data);

  // コンストラクタ宣言
  SC_CTOR( fir_filter )
  {
    SC_CTHREAD(thread0, clk.pos() );
    watching(reset.delayed() == false);
  }
};

#endif // _FIR_FILTER_H_
```

リスト2　FIRフィルタのメイン・スレッド（fir_filter.cc）

```
#include "fir_filter.h"

void fir_filter::thread0() {

UINT8  data;
UINT8  result;
UINT8  mode;

/* リセット定義 :
   CYN_PROTOCOLはサイクル・アキュレート・ブロックを
   指定するForte社Cynthesizerの属性  */
{ CYN_PROTOCOL( "Reset" );
     input_ready = 0;
     output_valid = 0;
     output_data = 0;
     for (int i=0; i<TAPS; i++) {
         shift[i] = 0;
     }
     wait();
 }

 sum_of_coefs = 0;
 for (int i=0; i < TAPS; i++) {
     sum_of_coefs += coefs[i];
 }

 // メイン・ループ
 while(1) {

     // 入力プロトコル
     { CYN_PROTOCOL("Read");
         input_ready = 1;
         do {wait(); } while(!input_valid.read());
         data = input_data.read();
         mode = input_mode.read();
         input_ready = 0;
     }

     // メイン関数の呼び出し
     if (mode == true)
         init(data);
     else
         result = calc_fir_filter(data);

     // 出力プロトコル
     { CYN_PROTOCOL("Write");
         output_valid = 1;
         output_data = result;
         wait();
         output_valid = 0;
     }
 }
};

// 初期化関数の定義
void fir_filter::init(UINT8 data)
{
 for (int i=0; i < TAPS; i++) {
      shift[i] = data;
  }
}

// メイン関数の定義
UINT8 fir_filter::calc_fir_filter(UINT8 data)
{
  UINT16 tmp = 0;  // sc_uint<16>

  for (int i=TAPS-1; i > 0; i--) {
      shift[i] = shift[i-1];
  }
  shift[0] = data;

  for (int i=0; i < TAPS; i++) {
     tmp += shift[i] * coefs[i];
  }

  return (tmp/sum_of_coefs);
}
```

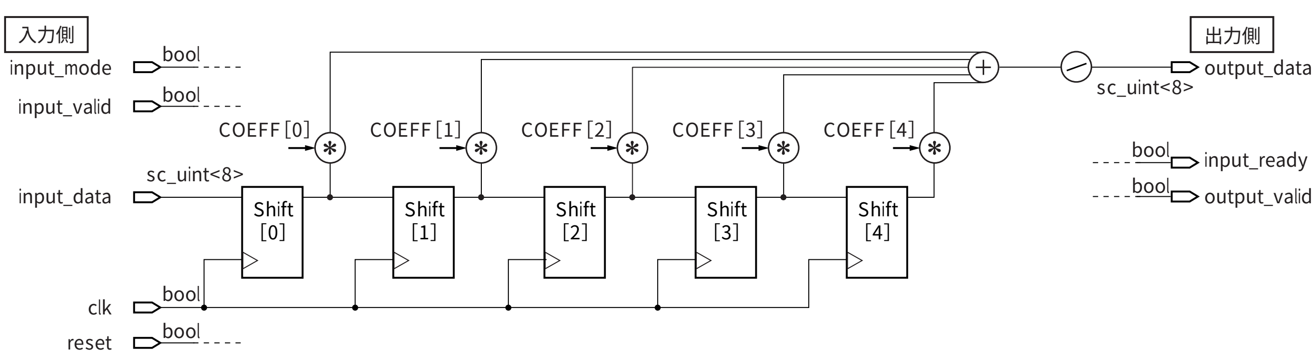


図1　FIRフィルタの構造

リスト1，リスト2に示したFIRフィルタの構造を示す．記述例は5タップのFIRフィルタで，乗算の係数はデータ型宣言時に初期値として与えられている（デフォルトではROMにマッピングされる）．レジスタは配列宣言されており，デフォルトではRAMにマッピングされる．このフィルタは，ハンド・シェイキングで外部との同期をとる．

イルに分割してもどちらでもかまいません．通常，SC_MODULE宣言部をヘッダ・ファイルとしてファイル属性.hを付けて作成し，メイン・スレッドのアルゴリズム記述は，.cpp，.ccというファイル属性を付けて作成します．

ヘッダ・ファイルは，同じファイルが複数回読み込まれるのを防ぐために#ifndef文を用います．一度，ファイルを読み込んでマクロが定義されていれば，重複して読み込まれることはありません．ヘッダ・ファイルの最初には，以下のようにインクルード文を用いて，SystemCを定義しているsystemc.hファイルを読み込みます．

```
#include <systemc.h>
```

また，必要があれば，さらに別のヘッダ・ファイルも読み込みます．

表1　ビヘイビア合成でサポートするC/C++とSystemCの標準データ・タイプ

bool	ブール(true，false)
char	文字(キャラクタ)
short	16ビット整数
int	32ビット整数
long	64ビット整数
double	64ビット浮動小数点
unsigned short	16ビット符号なし整数
unsigned int	32ビット符号なし整数
unsigned long	64ビット符号なし整数

(**a**) C/C++の標準データ型

sc_int<N>	Nビット整数
sc_uint<N>	Nビット符号なし整数
sc_bit	‘1’，‘0’の2値
sc_logic	‘1’，‘0’，‘x’，‘z’の4値
sc_bv<N>	sc_bitのNビット・ベクタ
sc_lv<N>	sc_logicのNビット・ベクタ
sc_bigint<N>	sc_intの64ビット以上
sc_biguint<N>	sc_uintの64ビット以上
sc_fixed<NW, NI>	固定小数点
sc_ufixed<NW, NI>	符号なし固定小数点

(**b**) SystemCの標準データ型

リスト1の6行目の以下の記述ではテキスト・マクロを定義しており，記述内のTAPS文字が値5に置き換わります．

```
#define  TAPS   5
```

リスト1では，SC_MODULEを用いてモジュールを定義しています(SC_MODULEについては本誌2004年12月号，pp.115-120の連載第2回で説明したので，ここではとくに説明しない)．

記述はフリー・フォーマットで，読みやすくするためにタブや空白，改行を自由に入れることができます．使用できる文字は，アルファベット，数字，アンダ・スコア(_)です．大文字と小文字は別の文字として扱われます．

コメントについては，1行のコメントは // で表し，改行までがコメントとして扱われます．複数行に渡るコメントには /*　*/ を用います．記述の管理や可読性を向上させるため，積極的にコメントを入れてください．

● ハードウェア設計向けのデータ・タイプを用意

表1に，ビヘイビア合成ツールがサポートするC++データ・タイプおよびSystemCデータ・タイプを示します．

1) 整数データ・タイプ

ビヘイビア合成でサポートされている整数のC/C++データ・タイプは，bool，char，short，int，longなどで，これらのデータ・タイプは符号付き整数の変数を宣言します．また，符号なしの場合，unsigned int，unsigned shortのようにunsignedを付加します．

表2　C/C++データ・タイプに対するSystemCデータ・タイプのサイズ
サイズについて，char≦short≦int≦longの関係を保持しなければならない．

C++データ・タイプ	ビヘイビア合成におけるサイズの割り付け
bool	sc_int<2> 注
char	sc_int<8>
unsigned char	sc_uint<8>
short	sc_int<16>
unsigned short	sc_uint<16>
int	sc_int<32>
unsigned int	sc_uint<32>
long	sc_int<32> または sc_int<64>
unsigned long	sc_uint<32> または sc_uint<64>
long long	sc_int<64>
unsigned long long	sc_uint<64>

注：boolタイプはC++では符号付きのように扱われるため，デフォルトでは2ビットにマッピングされる

SystemCのデータ・タイプでは，符号付き整数sc_int<>，符号なし整数sc_uint<>，ビット・ベクタsc_bv<>，論理タイプsc_logicなどのデータ・タイプをサポートしています．sc_int<>やsc_uint<>の最大幅は64ビットで，それ以上のビット幅が必要な場合には，sc_bigint<>やsc_biguint<>を用います．

数値は，数字だけの場合には10進数を意味します．16進数に対しては，0xff，0x12abのように0xを付加し，8進数に対しては，075，0246のように数値の最初に0を付加します．

ハードウェア化の際には，実際のビット幅を確定する必要があるため，C/C++データ・タイプは，sc_int<>またはsc_uint<>で示されるビット幅にマッピングされます．C/C++データ・タイプとそれに対応するSystemCデータ・タイプを**表2**に示します．

整数データ・タイプの場合，特定の1ビットを選択するには [ ] を用いてビットを指定し，複数のビットをレンジ選択するには ( ) または .range()を用います．また，複数の変数の連接には ( ) を用います．ビットの扱いは，例えば**リスト3**のようになります．

2) 固定小数点データ・タイプ

ビヘイビア合成では，符号付き固定小数点sc_fixed<>，符号なし固定小数点sc_ufixed<>を使用す

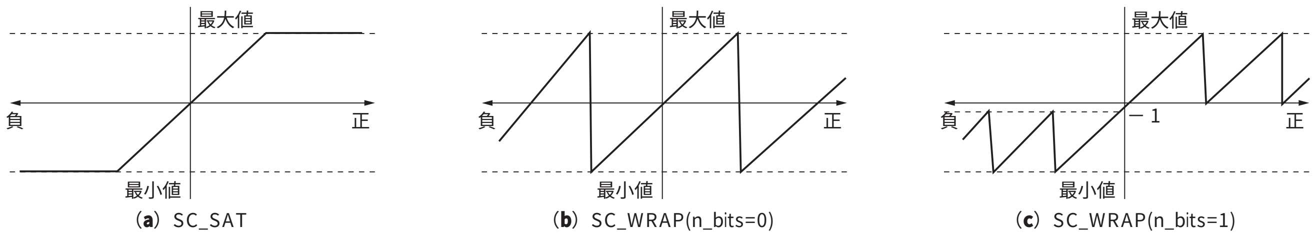


図2 sc_fixedにおけるオーバフロー動作
SC_SATは最大値または最小値に飽和．SC_WRAPではカウンタのように最小値に戻るが，n_bits＝0で正の場合は最小値に戻り，n_bits＝1では符号を保って‘0’に戻る．n_bitsで飽和するMSBからのビット数を示す．

リスト3 整数データ・タイプのビットの扱い

```
sc_uint<32>  a;
sc_uint<16>  b, c;
sc_uint<1>   d;

b = a.range(31,16);      // レンジ選択
c = a.range(15,0);
d = b[0];                // ビット選択
a = (b, c);              // 連接
```

ることができます．構文は次のようになります．

```
sc_fixed<wl, iwl, q_mode, o_mode,
          n_bits>  オブジェクト名, … ;
sc_ufixed<wl, iwl, q_mode, o_mode,
          n_bits>  オブジェクト名, … ;
```

ここで，wlは値の表現に必要な総ビット数，iwlは整数部の表現に必要な小数点より左側のビット数を示します．q_modeは量子化モードで，最下位ビットの精度を超える場合の動作を指定します．例えば，SC_TRN（デフォルト）は切り捨て，SC_RNDは四捨五入を行います．o_modeはオーバフロー時の動作を指定し，SC_SATはオーバフローの際に最大値または最小値に飽和します（**図2**）．また，SC_WRAP（デフォルト）はカウンタのようなラップ・アラウンドを指定します．値が正でn_bits=0の場合，最大値でオーバフローした際には最小値に戻り，n_bits=1では‘0’に戻ります．n_bitsの指定によりMSB（most significant bit）から何ビットが飽和するかを指定します．

3）配列

データ・タイプ宣言に対して，さらにサイズを指定することで多次元配列を宣言することができます．例えば，2次元配列，3次元配列などを使用できます．**リスト1**の26行目では，以下のような記述によって8ビット幅，ワード数5の配列を宣言しています．

```
UINT8  shift[TAPS];
```

また，**リスト1**の10行目では以下のようにconst文の宣言とともに配列に初期値を代入しています．

```
const sc_uint<4> coefs[TAPS]
                 = {1, 2, 3, 2, 1};
```

ビヘイビア合成では，通常，配列はRAMやROMとして実現しますが，配列に対して展開を指示することで，RAMのそれぞれのビットをレジスタとして実現したり，ROMを組み合わせ回路として実現したりすることができます．配列をRAM/ROMとして実現した場合，一度にアクセスできるのは一つのインデックス（アドレス）で指定したデータに限られます．一方，レジスタに展開すれば，同時に複数のインデックスの配列にアクセスできるため，処理サイクル数を短くすることができます（**図3**）．

ビヘイビア合成では，ソフトウェアでよく用いられる，ポインタを用いた配列のダイナミックな（動的な）宣言を使用することはできません．配列はかならずサイズを指定してスタティックに（静的に）宣言することが必要です．そのため，ビヘイビア合成の利用を意識してアルゴリズムを開発する場合，ダイナミックな配列の使用を制限する必要があります（ソフトウェア開発者は，配列のサイズを意識せずに使用することが少なくない．そのため，アルゴリズム記述を合成する際に配列サイズを確定する作業が必要となり，余分な手間がかかる場合がある）．

4）ユーザ定義型データ・タイプ

既存のデータ・タイプに対して，typedef文を用いて別の名まえを付けることができます．**リスト1**の7行目では，以下の記述によって，sc_uint<8>に対して別名UINT8を与えています．

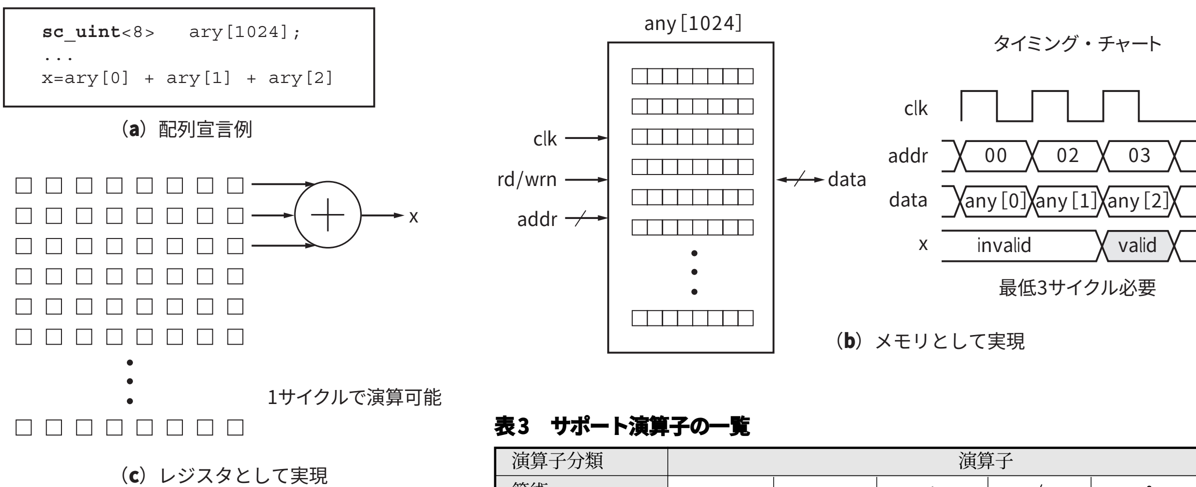


図3 配列のメモリ化とレジスタへの展開

配列は，メモリにマッピングするかレジスタに展開する．メモリにマッピングする場合，複数サイクルのアクセスが必要．レジスタに展開した場合には，一度に複数のデータにアクセスできる．

表3 サポート演算子の一覧

演算子分類	演算子					
算術	+	-	*	/	%	
代入演算子	+=	-=	*=	/=	%=	
単項演算子	+	-				
インクリメント/デクリメント	前置++	後置++	前置--	後置--		
ビット演算子	&	|	^			
代入演算子	&=	|=	^=			
単項演算子	~					
関係演算子	==	!=	<	<=	>	>=
シフト演算子	>>	<<	>>=	<<=		
ビット選択	[x]					
パート選択	(i,j)					
連接	(,)					
C整数への変換	to_int	to_long	to_int64	to_uint	to_uint64	to_ulong
代入	=					

注：データ・タイプによってサポートしている演算子が異なる．

リスト4 データ・タイプの変換

```
sc_int<16> a;
sc_uint<16> b;
sc_uint<32> c;

b = (sc_uint<16>)(a+1);
c = (sc_uint<32>)((sc_uint<16>)a+b)
```

```
typedef sc_uint<8>  UINT8;
```

5）タイプ変換

算術演算や関数の呼び出し時には，暗黙のタイプ変換が行われます．例えば，2項演算で被演算子のタイプが異なる場合，ビット幅の大きいタイプに変換します．また，代入文の右辺は左辺のタイプに合わせてから代入します．

暗黙的な変換ではなく，キャスト演算を用いて明示的にタイプを変換することもできます．例えば，符号付きから符号なし，符号なしから符号付きのようにデータ・タイプを変換します．その例を**リスト4**に示します．

6）列挙型

リスト1の記述例では使用していませんが，列挙型タイプenumもサポートしています．例えば，以下のように定義できます．

```
enum week = {MON, TUE, WED, THU,
                    FRI, SAT, SUN};
```

● 演算子利用の可・不可はデータ・タイプによって異なる

表3にサポートしている演算子の一覧を示します．サポートしている演算子には，加減乗除などの算術演算子，関係演算子，論理演算子，ビット演算子，代入演算子などがあります．C/C++の整数タイプについてはすべての演算子を使用できますが，それ以外については，SystemCではデータ・タイプによって使用できる演算子と使用できない演算子があるので注意してください．

ビヘイビア合成では，演算子は，被演算子のビット幅に対応した論理レベルの演算リソースにバインディングされます．演算リソースは，通常，すでに用意された複数の演算リソースからタイミング情報に基づいて，クロック周期を満足する範囲で最小の面積を持つリソースが選択されま

す(本誌2005年2月号，pp.138-143を参照).

なお，通常，2のべき乗の乗算や除算はビット・シフトとして実現します．

● ループ文は並列化やパイプライン化の指定で高速処理

繰り返して実行する記述に対して，ループ文を用いて繰り返しを定義することができます．ループ文としては，for文，while文，do-while文をサポートしています．

for文は，インデックス変数を用いて，ループを実行する条件とインデックス変数のインクリメント/デクリメントを指定します．while，do-whileでは，指定した条件が真(true)であればループを実行し，偽(false)の場合はループを実行しません．

リスト2の33行目では，do-while文として次のような記述を用いています．

```
do {wait(); }
            while(!input_valid.read());
```

この構文は，入力信号input_validの値が‘1’(真)になるまで待ち，‘1’になったら次の記述に進みます．この記述はSC_CTHREADで使用可能な以下の記述と同じです．

```
wait_until(input_valid.delayed()==
                                  true);
```

なお，同じような記述を，以下のようにwhile文でも記述することができます．

```
while (!input_valid.read())
                                  {wait()};
```

do-while文ではループ内の構文を一度実行してから条件を判断するのに対して，while文ではまず条件を判断してからループに入るかどうかを決めます．そのため，while文では最初の条件判断が偽(‘0’)であれば，ループ内の構文は実行されません．

ループ文は繰り返し構文であるため，ビヘイビア合成は記述どおりにループ内の構文をインデックスごとに毎回実行する回路を生成します．そのため，例えばループ回数が4回でループ内の構文を実行するのに4サイクル必要であるとすると，ループの実行には4回×4サイクルの合計16サイクルが必要となります(**図4**)．

ループに対してインデックスごとの処理が独立していれば，ループ内の構文の並列処理を指定することでサイクル数を短縮できます．しかし，ハードウェアはインデックスごとに複製されるので，必要なハードウェア量は増加します．

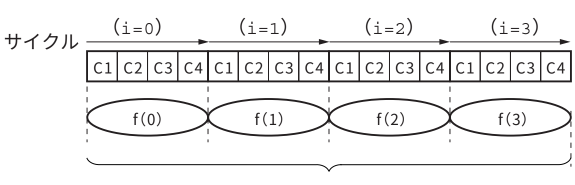
また，パイプライン化を指定することで，処理を開始するサイクルをずらしながら各インデックスの処理を実行させることができます．**図4(d)**のように実行開始が1サイクル間隔であれば，3＋4の合計7サイクルで実行できます．要求されるRTL回路の性能に従って，順序的な実行，並列処理，パイプライン処理を指定します[注]．

break文，continue文によって，ループの実行を中断

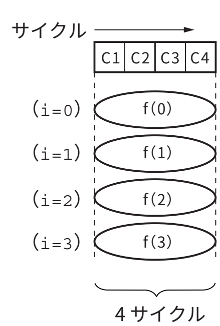
注：ただし，このような機能を持つかどうかはビヘイビア合成ツールに依存する．

```
for (int i = 0; i<4; i++ ) {
 z[i] = x[i] * y[i];
....
}
```

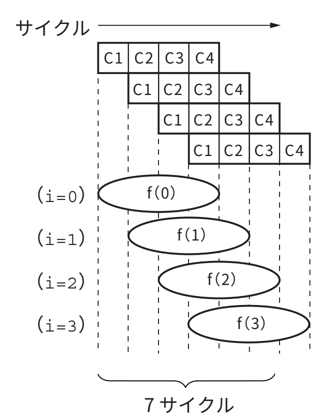
(**a**) forループ文



(**b**) 順序的処理



(**c**) 並列化



(**d**) パイプライン化

図4　ループの並列化とパイプライン化

展開していないループでは，同じハードウェアを何度も使用するため，必要な演算リソースは少なくなるが，ループ演算の完了に長いレイテンシを要する．一方，並列化やパイプライン化を指定するとリソース数は増えるが，必要なレイテンシは短くなる．

させることができます．break 文では，ループ全体を抜けて，実行をループ文の後に移します．continue 文では，continue 文に続く構文のループ内の処理を中断し，次の繰り返し処理を実行します．

● 条件文はステート・マシンまたはセレクタに置き換わる

条件文には，if 文，if-else 文，switch 文があります（**リスト2**の40行目では if 文を用いている）．また，条件演算子 ?: も使用することができます．今回の記述例では switch 文を使用していませんが，switch 文を使用することで，例えば，**リスト5**のような分岐を定義することができます．

各分岐の最後には，上述の break 文を指定します．break 文がないと，次の分岐も実行してしまいます．

ビヘイビア合成では，条件分岐はステート・マシンの一部として実現するか，データパス・コンポーネント（部品）のセレクタとして実現します（**図5**）．

ステート・マシンの場合，条件判断を行ってどの状態（ステート）に進むかを判断し，次のサイクルでその状態に進みます．そのため，条件分岐の実行に1サイクルが必要となります．

セレクタの場合，条件分岐に対しデータパス部を構成するセレクタ・コンポーネントとして実現するので，条件判断に余分なサイクルを必要としません．その結果，実行サイクル数が短くなります．ただし，セレクタ・コンポーネントの分だけ，面積が大きくなる可能性があります．

リスト5　switch文を使用した条件分岐

```
switch(mode) {
  case 0 : y = a; break;
  case 1 : y = b; break;
  case 2 : y = a + b; break;
  case 3 : y = a - b; break;
  default : y = 0;
}
```

● 関数を利用して記述を構造化

関数化を行うことで，繰り返して使用する記述を効率化したり，記述を構造化して処理フローをわかりやすくすることができます．**リスト1**の30～32行目では，三つの関数をメンバ関数としてあらかじめ宣言しています．関数本体の記述は後で定義しています．C/C++では，関数をあらかじめ宣言して，本体を後から定義することができます．二つの関数 thread() と init() は戻り値がないので，void（無効）を指定しています．

SC_MODULE 内で宣言しているメンバ変数（**リスト1**の26，27行目）については，直接，関数内から値を参照したり，値を書き込んだりすることができます．一方，メイン・スレッド内に宣言されたローカル変数（**リスト2**の5，6行目）は，引き数として関数に渡し，return 文によって値を戻しています．

関数から複数の値を戻すためには，ポインタによるアドレス渡しまたは参照渡しによって値を戻します．ポインタを用いた場合，calc_fir_filter 関数は**リスト6**のように定義できます．

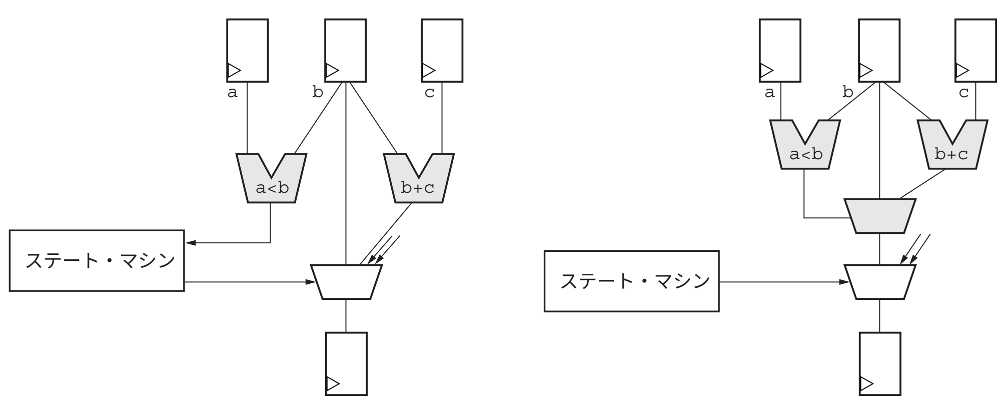
この場合，上記の関数を呼び出す際には，以下の記述のように変数のアドレスを指定します．

```
calc_fir_filter(&result, data):
```

また，calc_fir_filter 関数は，参照を用いて**リスト7**のように記述することもできます．この場合の関数呼び出しは，次のようになります．

```
if (a<b) z=b;
else     z=b+c;
```

（**a**）記述例



（**b**）分岐をステート・マシンで制御

（**c**）分岐をセレクタとしてコンポーネント化

図5
条件文の実現
条件分岐の結果をステート・マシン（FSM）の状態の遷移として実現するか，セレクタ（データパス・コンポーネント）として実現するかを選択することができる．

リスト6　ポインタを用いた場合のcalc_fir_filter関数

```
void fir_filter::calc_fir_filter
      (UINT8 *result, UINT8 data)
{
    . . .
    *result = tmp/sum_of_coefs;
}
```

リスト7　参照を用いた場合のcalc_fir_filter関数

```
void fir_filter::calc_fir_filter
     (UINT8 &result, UINT8 data)
{
    . . .
    result = tmp/sum_of_coefs;
}
```

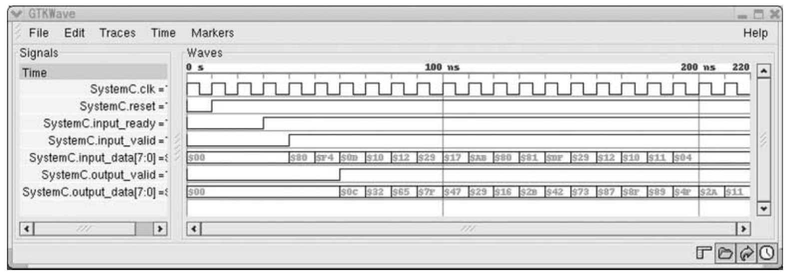


図6　FIRフィルタの入出力波形

FIRフィルタをパイプライン化してビヘイビア合成を実行した後のRTL設計の入出力波形を示す．加算と乗算の実行に1サイクル，除算の実行に1サイクルかかっているため，出力は2サイクル遅れとなる．演算回路系の設計ではパイプライン化は重要．

```
calc_fir_filter(result, data):
```

ビヘイビア合成で使用可能なポインタは，関数から値をアドレス渡しとして戻すときにのみ使用可能であると思っていたほうが安全です．ビヘイビア合成におけるポインタの使用は制限されているので注意してください．

● ビヘイビア合成を利用すると仕様の変更が容易に

フィルタでは，クロックごとに連続的にデータを入力し，クロックごとに連続的にデータを出力する必要があります．そこで，今回の記述例に対して，二つの配列の展開，すべてのループの展開，およびメイン・ループに対してインターバル1のパイプライン化を指定して，ビヘイビア合成を行いました．その結果，4個の加算器，1個の乗算器（係数1，2の乗算は単なるシフトとして実現される），1個の除算器注を用いたRTL回路が生成されました．これは，**図1**に示した構造と同じです．ビヘイビア合成によって生成されたRTL記述のシミュレーション結果を**図6**に示します．

記述例は5タップのFIRフィルタなので，ビヘイビア合成を利用するほどの規模ではありません．しかし，タップ数や係数などの仕様の変更がRTL記述の場合よりも容易であることが，記述例からもわかると思います．メイン関数の行数が数百行以上の場合でも，基本的なビヘイビア記述の構造は同じです．

今回は，ビヘイビア合成可能な基本的なC/C++言語のデータ・タイプや演算子，ループ文，条件文について紹介しました．SystemC/C++を用いた記述は，HDL記述に比べて難しいと感じるかもしれませんが，慣れてしまえば，それほどでもありません．ビヘイビア合成を用いることで設計の抽象度が上がれば，仕様の変更がHDLの場合よりも容易になります．

注：通常は除算器を必要としないように係数を決める．

参考・引用*文献

(1) OSCI（Open SystemC Initiative）；SystemC Synthesizable Subset，2004.
(2) OSCI（Open SystemC Initiative）；SystemC version 2.0 User's Guide / FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0，2002.
(3) 長谷川裕恭，小川丈博；連載・SystemCを使ったシステム設計，Design Wave Magazine，2003年7月号～2004年7月号.
(4) Thorsten Grotker，Stan Liao，Grant Martin，Stuart Swan（柿本勝，河原林政道，長谷川隆 監訳）；SystemCによるシステム設計，丸善，2003年.
(5) 大村正之，深山正幸；C/C++によるVLSI設計，共立出版，2003年.

さくらい・たかし
フォルテ・デザイン・システムズ（株）

<筆者プロフィール>
桜井 至．EDA業界に入り，HDLとしてはVerilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はビヘイビア合成を担当．設計言語と合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」（テクノプレス社），および「Verilog-AMS入門」（CQ出版社）がある．現在，フォルテ・デザイン・システムズ（http://www.forteds.com/）に所属．